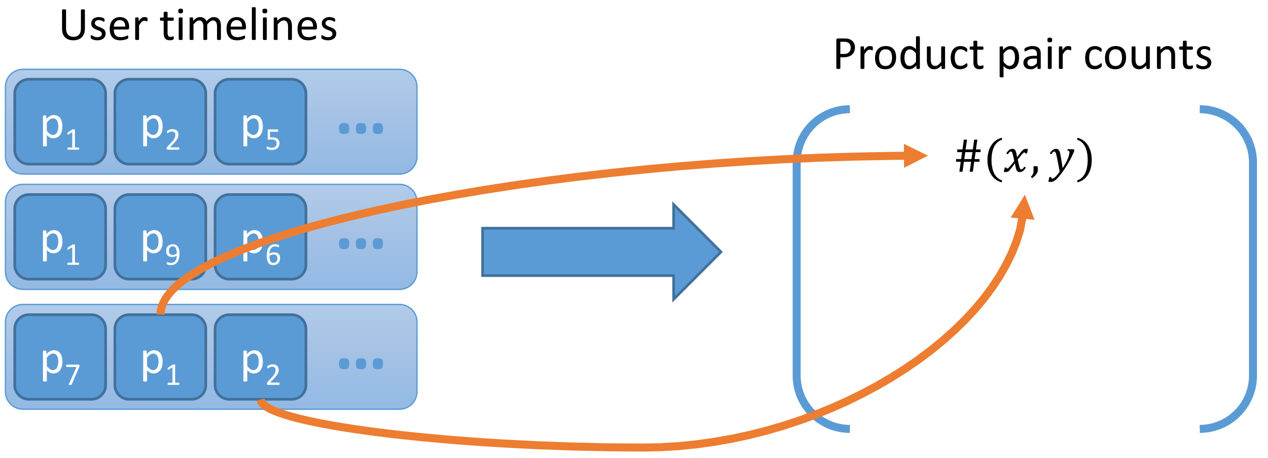
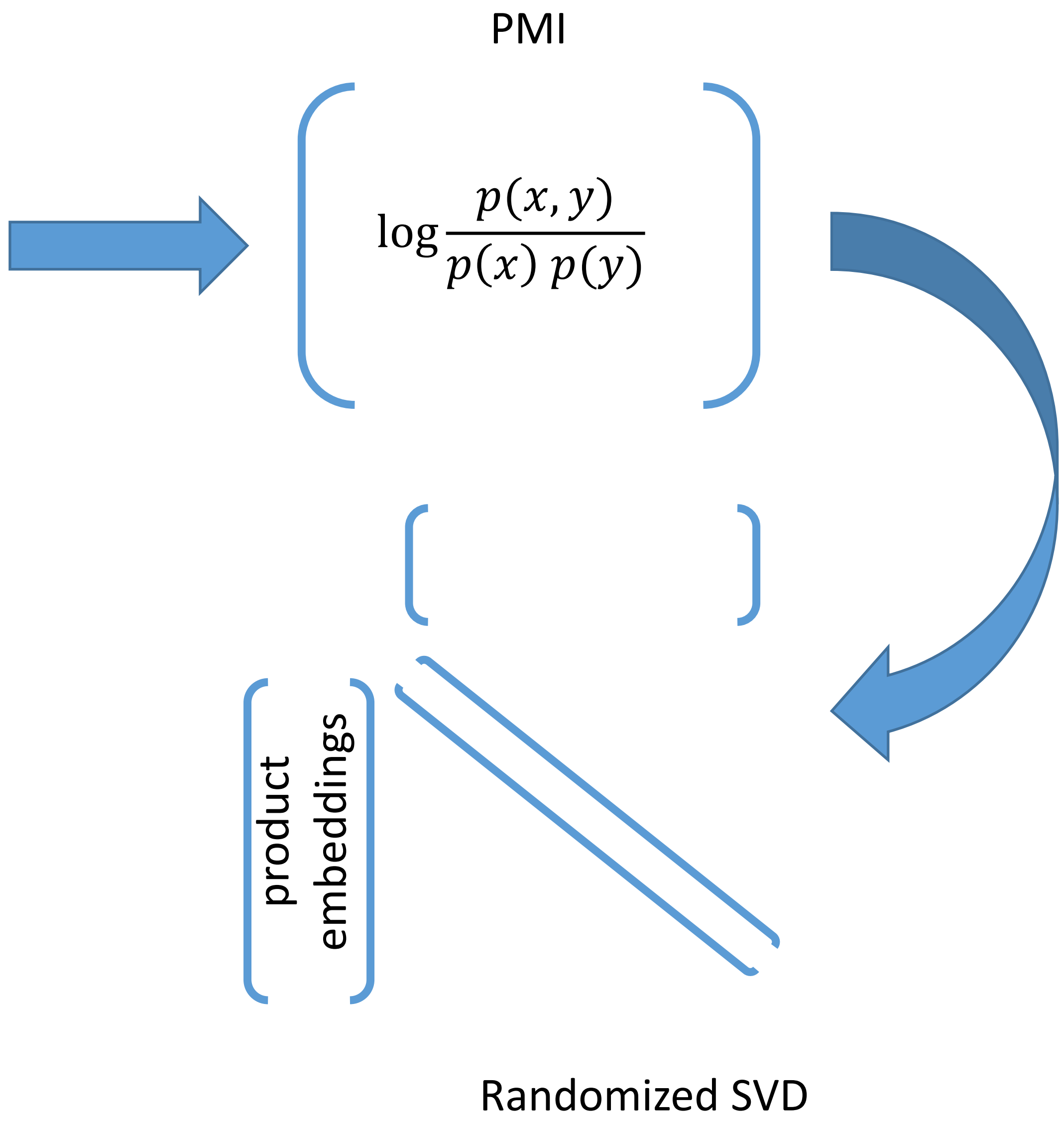
Part 2: Recommender systems in production

David Diebold
Martin Bompaire
Victor Paltz

Where did we stop ?

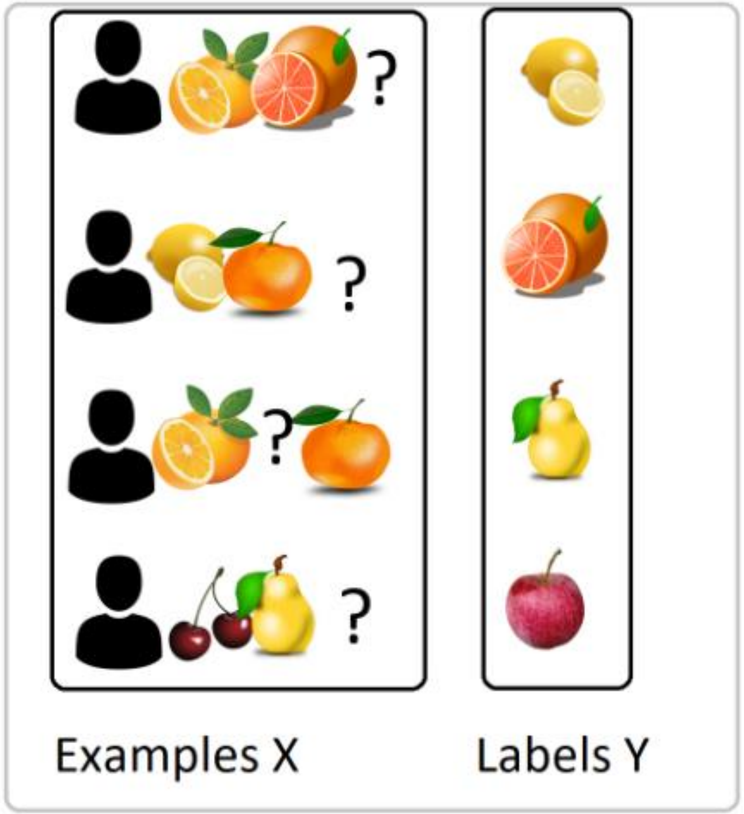




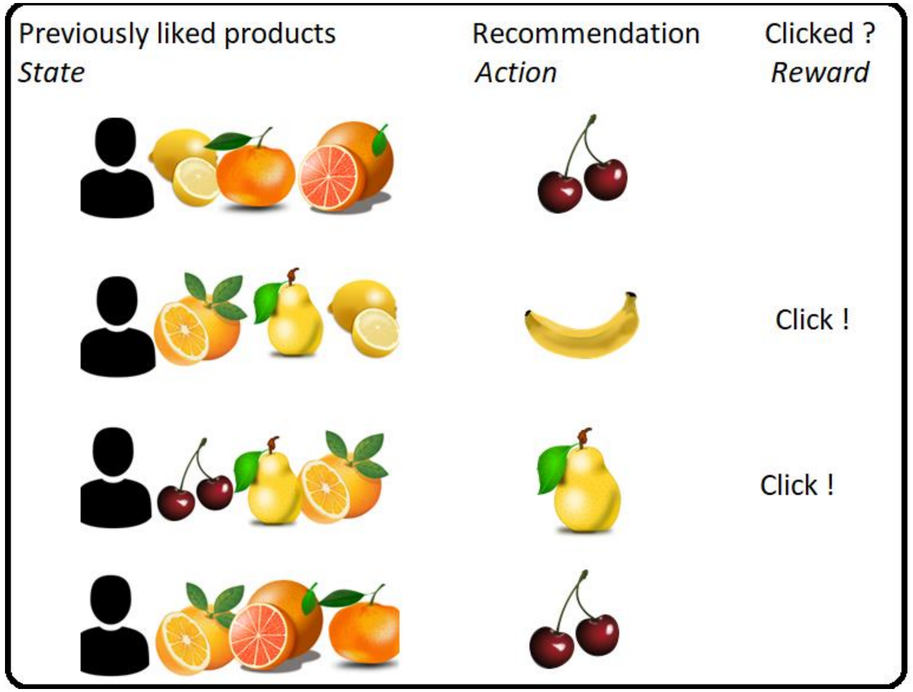
From an **explicit or implicit** user signal coming from user timelines, we can create **product embeddings** that encode **product affinity** through **cosine similarity** in a dense vectorial space

Bandit VS Organic

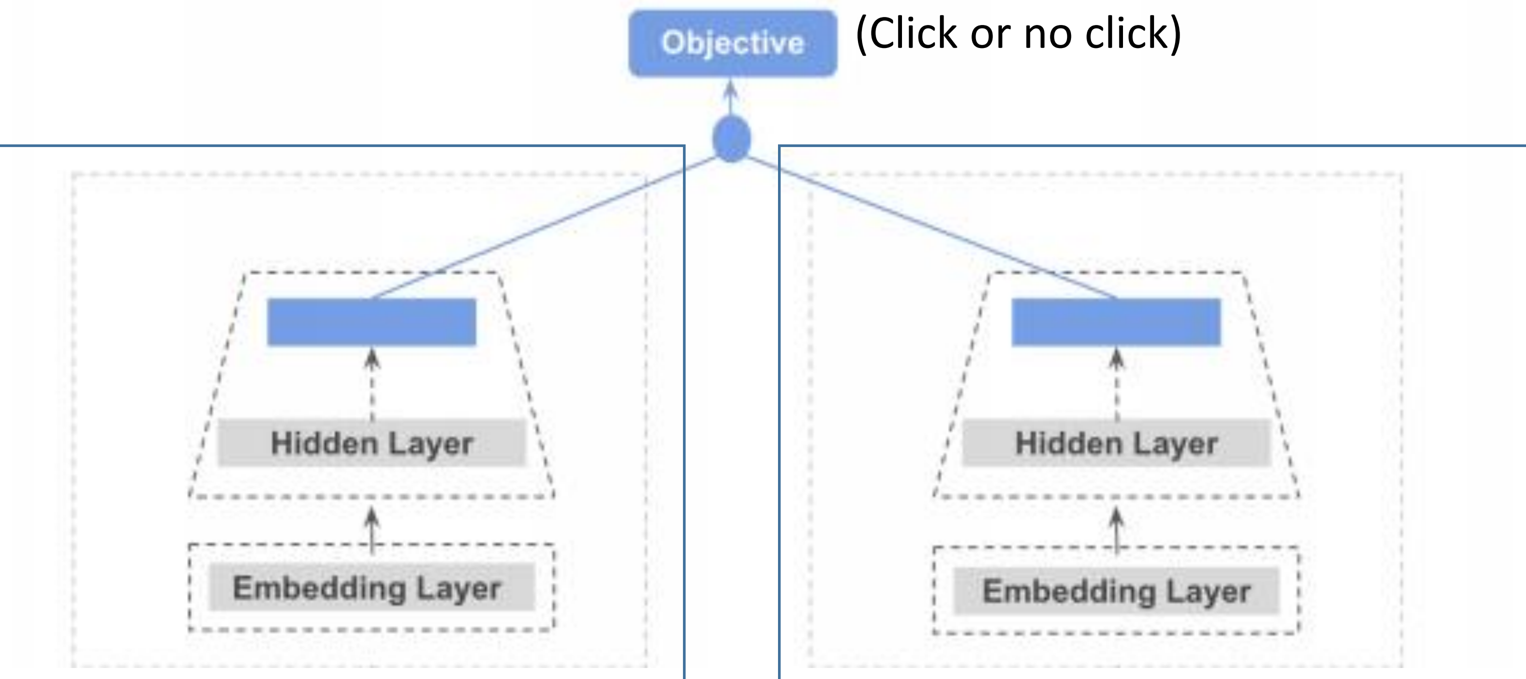
Organic



Bandit



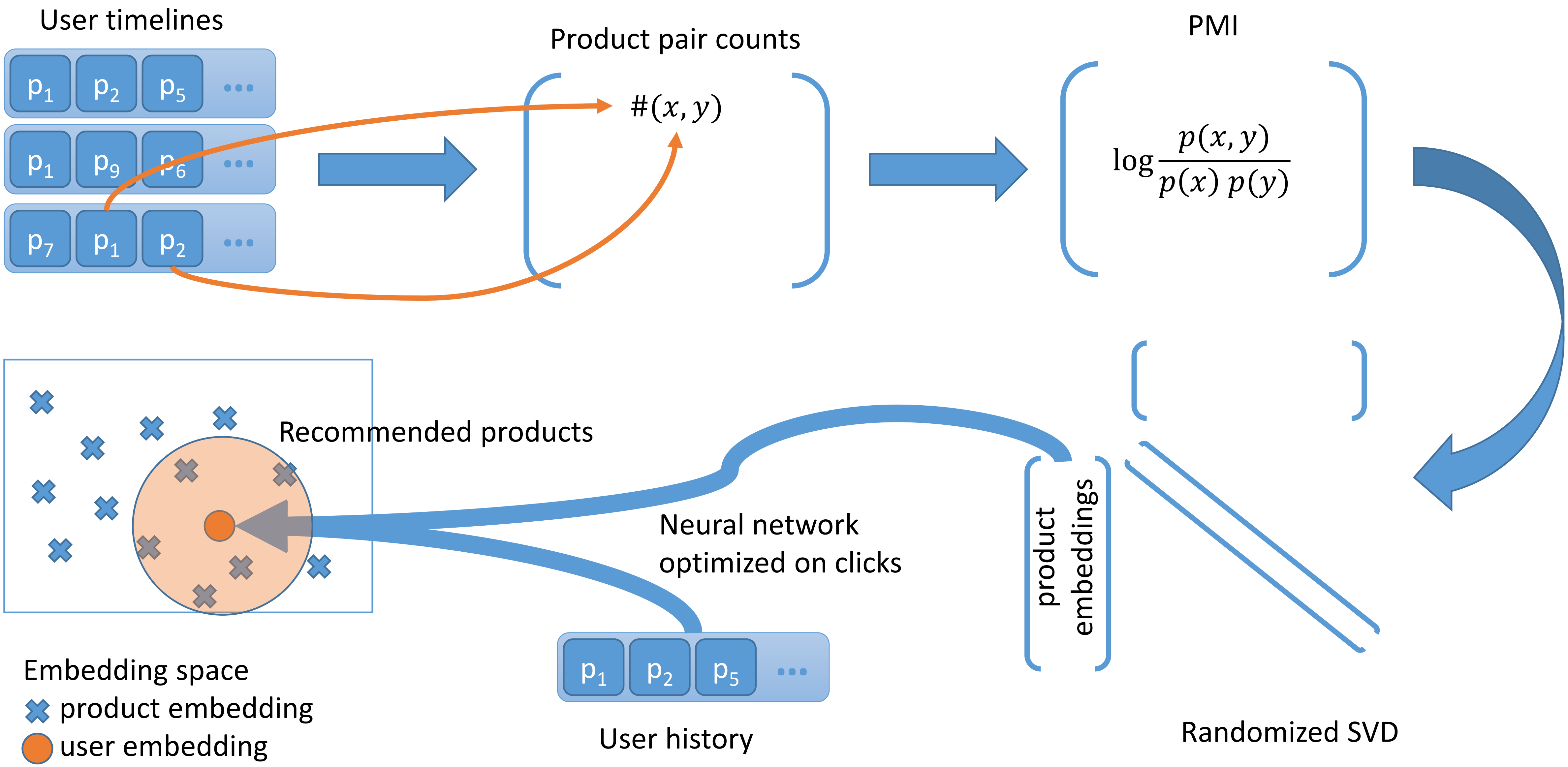
Deep learning model to learn the Bandit signal



Input: list of product embeddings seen by user. Model inferred online

Precompute offline the finetuned product embeddings and store it in a KNN

Where did we stop ?



Congrat! You can now build a powerful recommendation system! But…

- Will it be **fast** enough?
- Will it be **cheap** enough?
- Will it be **good** enough?
- Will it be **fast to rebuild** enough?

Solution:

Choose wisely the KNN algorithm and the DL model

KNN show time!

Overview

1. Brute-force
2. Tree based (KD-tree and ANNOY)
3. Hash based (LSH and Random projections)
4. Graph based (NSW and HNSW)
5. Inverted Index (IVF)
6. Quantized algorithms (IVFPQ)
7. KNN Fusion
8. KNN in practice! (Faiss and Autofaiss)
9. Bonus? (ScaNN, QUIPs)

1 – Brute Force

1. Brute force KNN

Time complexity for k=1

Algorithm	Average	Worst
Space	O(n)	O(n)
Search	O(n)	O(n)
Insert	O(1)	O(1)
Delete	O(1)	O(1)

Sometimes, brute force algorithms are just the best choice:

- Cheap to compute
- Fixed search time
- Exact search
- We can use any distance function (especially the ones that don't have the Triangle inequality properties such as the inner product)

Brute force KNN with numpy

```
# How to get the indices of the top k highest values per row :

import numpy as np

k = 20
array = np.random.rand(500, 500_000)

s1 = array.argsort()[:,-1:-k-1:-1]
s2 = np.argpartition(-array, range(k) )[:,:k]
s3 = np.argpartition(-array, range(k-1, 0, -1) )[:,:k]
s4 = np.argpartition(array, range(-k, 0))[:,-1:-k-1:-1]

print("argsort")
%timeit s1 = array.argsort()[:,-1:-k-1:-1]
print("argpartition v1")
%timeit s2 = np.argpartition(-array, range(k) )[:,:k]
print("argpartition v2")
%timeit s3 = np.argpartition(-array, range(k-1, 0, -1) )[:,:k]
print("argpartition v3")
%timeit s4 = np.argpartition(array, range(-k, 0))[:,-1:-k-1:-1]
```

```
argsort
30.3 s ± 172 ms per loop (mean ± std. dev. of 7 runs, 1 loop each)
argpartition v1
10.1 s ± 168 ms per loop (mean ± std. dev. of 7 runs, 1 loop each)
argpartition v2
9.98 s ± 27.1 ms per loop (mean ± std. dev. of 7 runs, 1 loop each)
argpartition v3
3.21 s ± 16.4 ms per loop (mean ± std. dev. of 7 runs, 1 loop each)
```

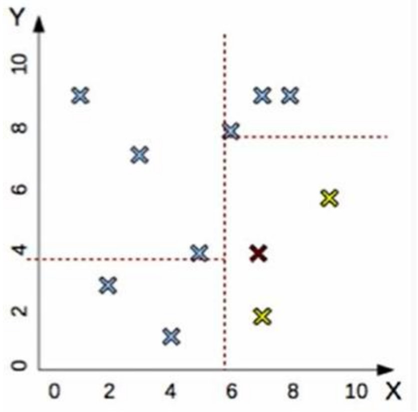
Using heapq.nlargest can be x10 slower, built-in vectorial numpy operations are the best

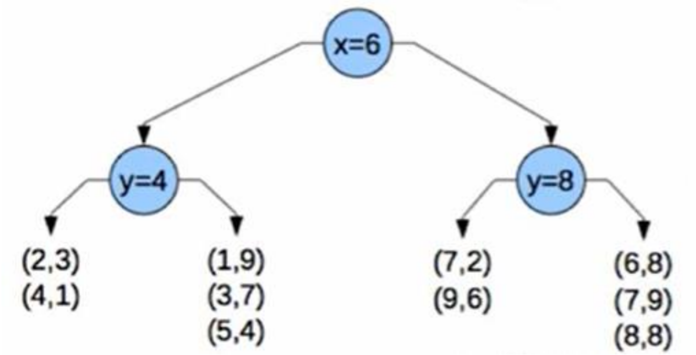
2- Hierarchical Trees

2. KD Tree

Time complexity for k=1

Algorithm	Average	Worst
Space	O(n)	O(n)
Search	O(log(n))	O(log(n))
Insert	O(log(n))	O(log(n))
Delete	O(log(n))	O(log(n))





Construction

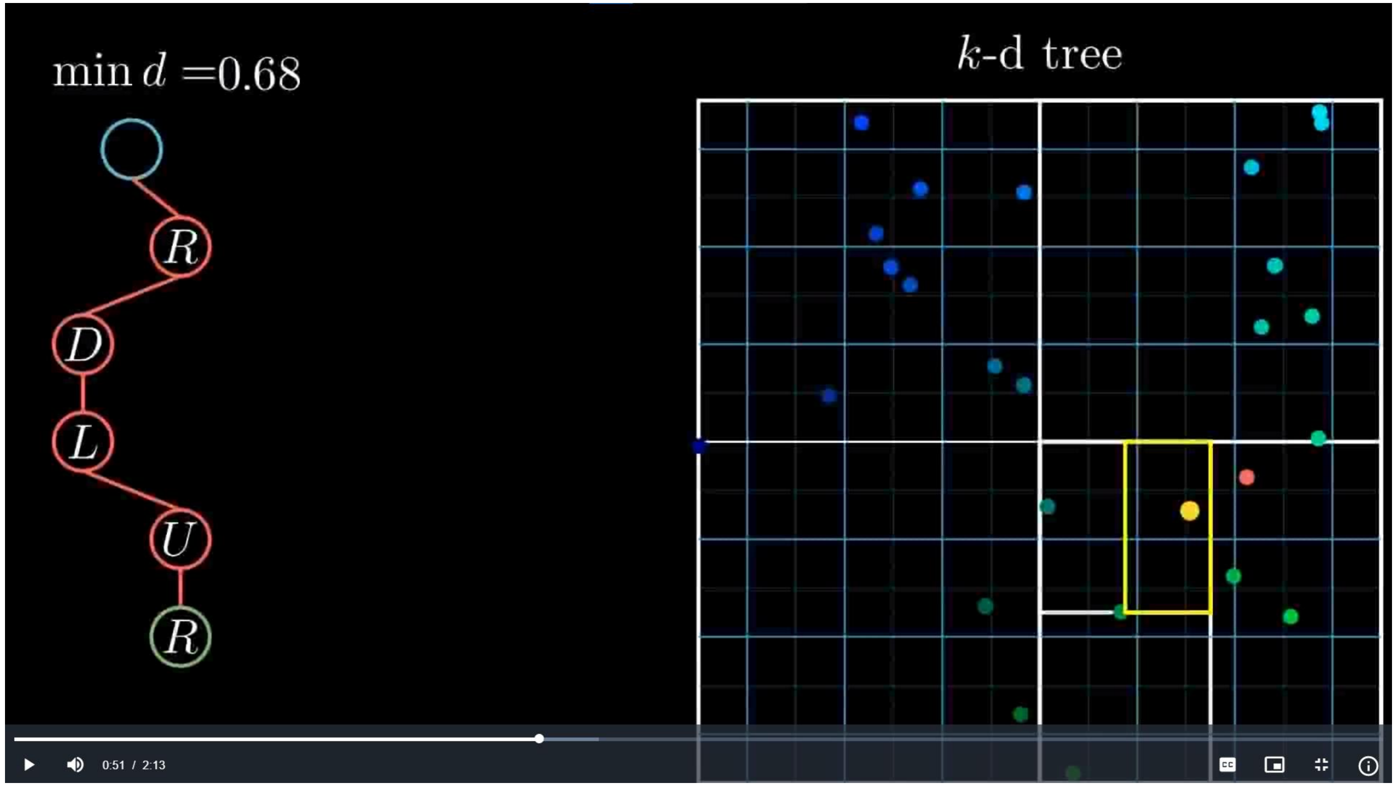
While cell contains 'too many' items:

- Pick random axis
- Find median
- Split cell

Search

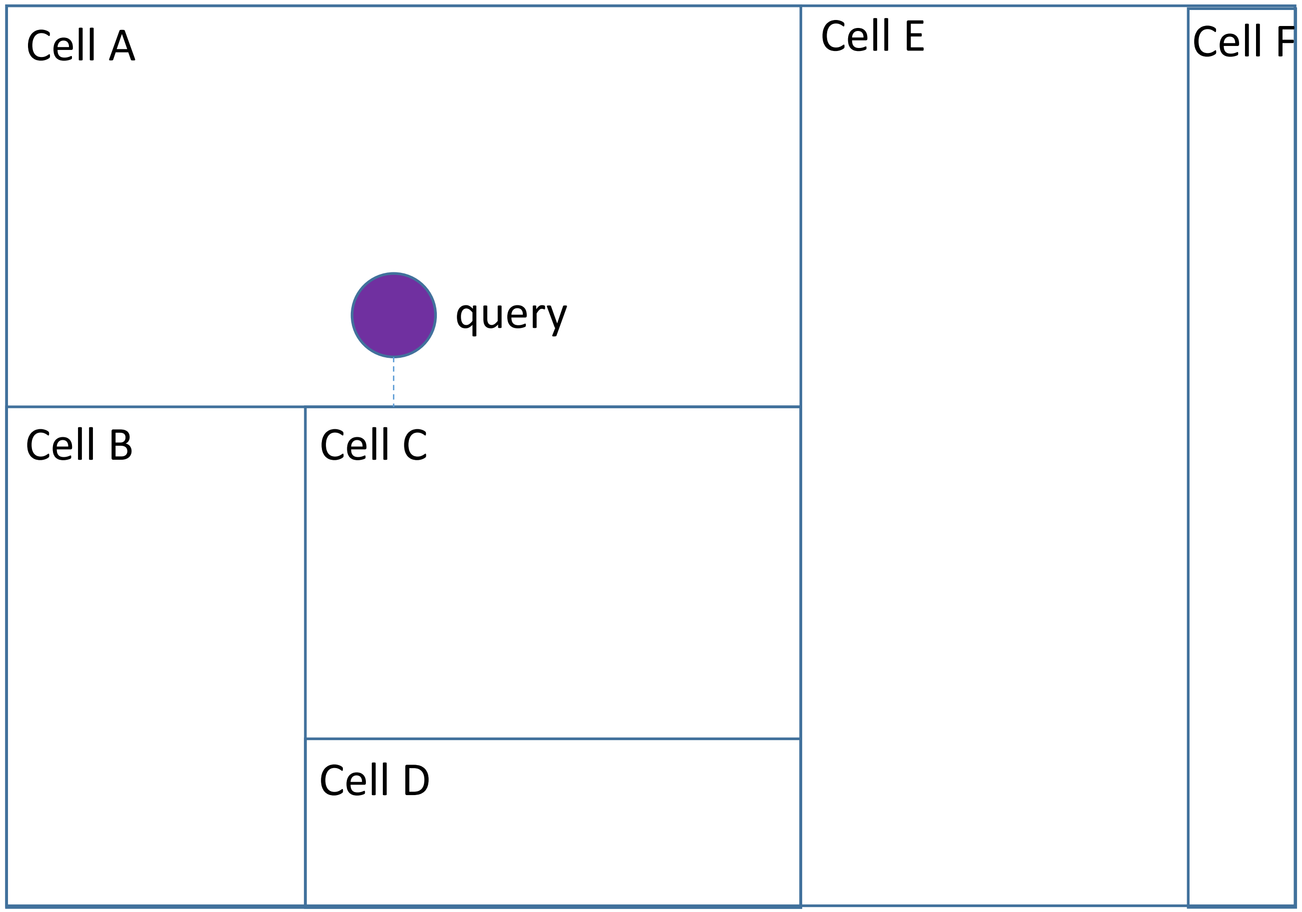
Naïve way: Walk the tree, then look for the nearest neighbor inside the leaf.

KD Tree: cool animation for exact seach



https://upload.wikimedia.org/wikipedia/commons/4/48/Kdtreeogg.ogv

KD Tree – Exact search



Ideas

It is fast to compute min distance between cell and queried point.

Don’t explore a cell when we know it’s min distance is larger than nearest point found so far.

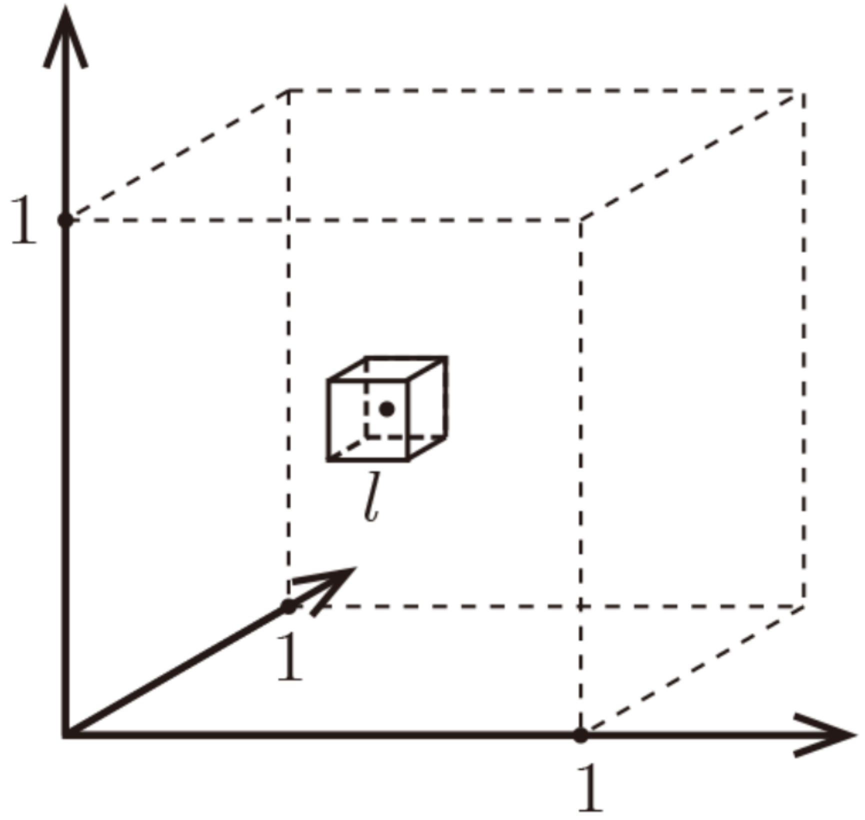
We use branch-and-bound to find nearest neighbors.

Start with nearest cells first to increase chances to minimize nearest point distance early.

Can be made approximate with some guarantees. Multiply d(query,cell) by (1+Epsilon)

Still, **not efficient with high dimension vector**

Curse of dimensionality



N points placed uniformly inside the space.

Let ℓ be the edge length of the smallest hyper-cube that contains all k-nearest neighbor of a test point.

Then $\ell^d \approx \frac{k}{n}$ and $\ell \approx \left(\frac{k}{n}\right)^{1/d}$. If $n = 1000$, how big is ℓ?

KD Tree - Curse of dimensionality

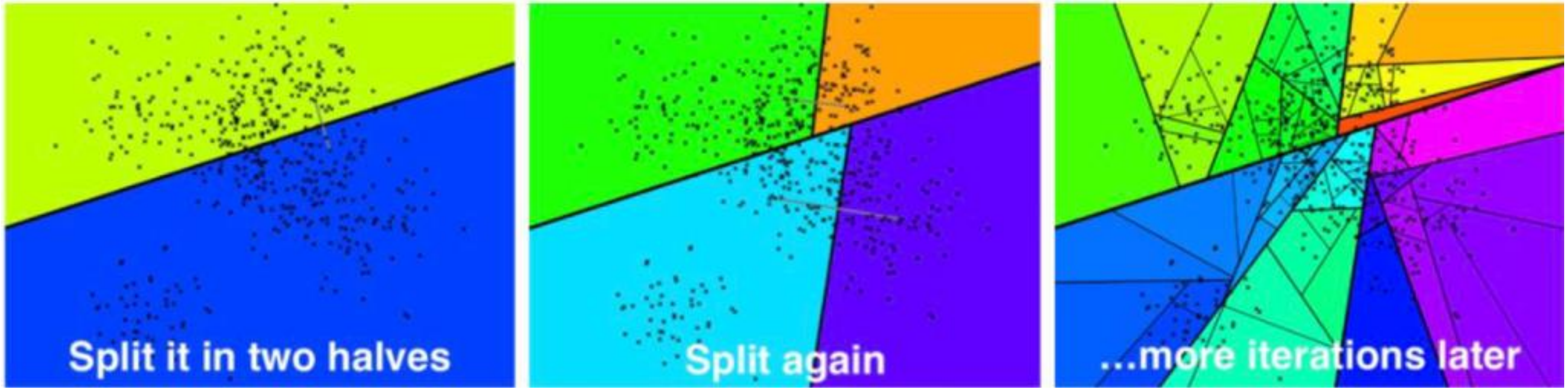
This effect complicates the nearest neighbor search in high dimensional space.

Indeed, it is not possible to quickly reject candidates by using the difference in one coordinate as a lower bound for a distance based on all the dimensions.

We need other algorithms compatible curse-of-dimension-friendly!

ANNOY

ANNOY (Approximate Nearest Neighbors Oh Yeah) is an algorithm based on random projections and trees. It was developed by Erik Bernhardsson in 2015 working at that time at Spotify. ANNOY is designed to search in date sets up to 100 to 1000 dense dimensions.



Annoy (Approximate Nearest neighbors Oh Yeah)

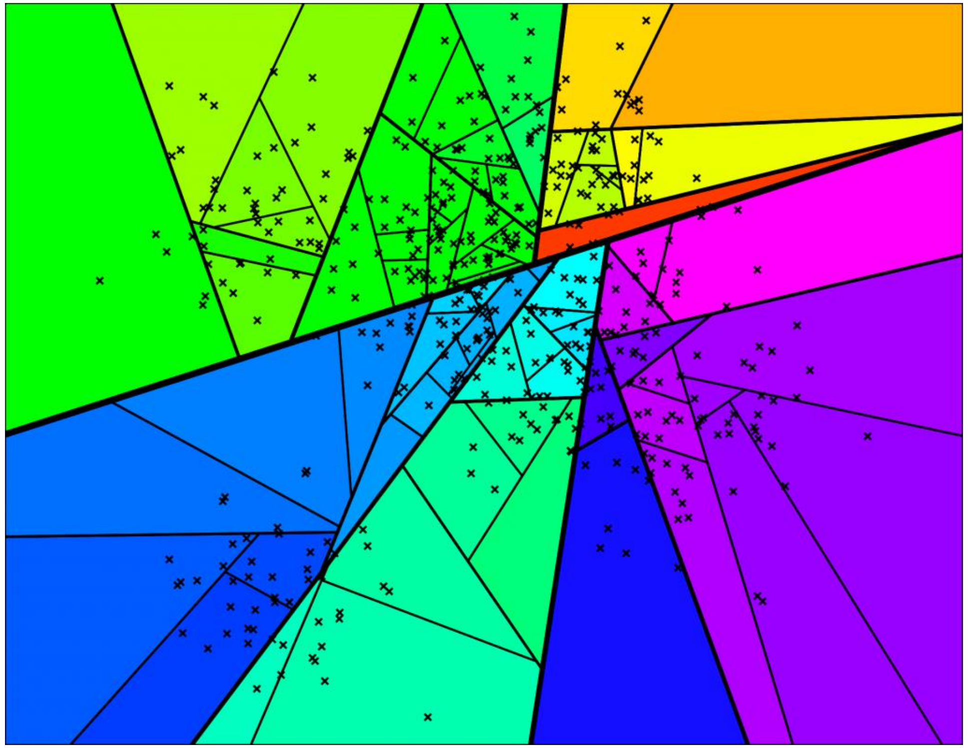
Construction

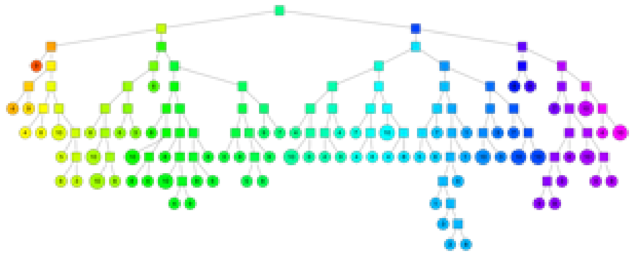
While cell contains 'too many' items:

- Pick two random points in cell
- Split along plane equidistant to the points

Create a Forest !

Great performances !





Search

Branch and bound with additional tricks.

Search on all trees at the same time using same priority queue.

Approximate: don't explore the other side if it's *too* far. Threshold can be changed during the search.

3- Hash based

Locality-sensitive hashing

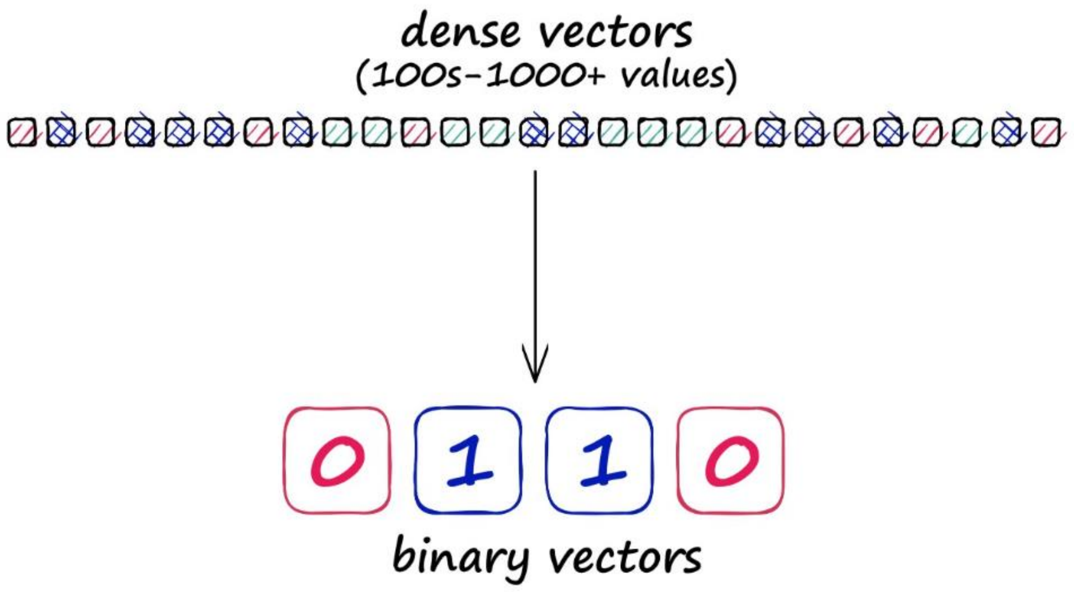
Here we want to find hash functions for which the hash is correlated to the distance of two vectors

1. Pr(h(a) == h(b)) is high if a and b are near
2. Pr(h(a) == h(b)) is low if a and b are far
3. Time complexity to identify close objects is sub-linear.

Locality-sensitive hashing

Performing search with LSH consists of three steps:

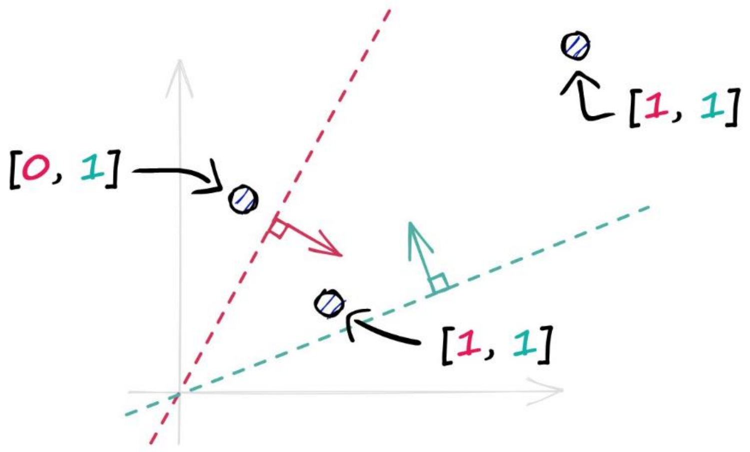
1. Index all of our vectors into their hashed vectors.
2. Introduce our query vector (search term). It is hashed using the same LSH function.
3. Compare our hashed query vector to all other hash buckets via Hamming distance — identifying the nearest.



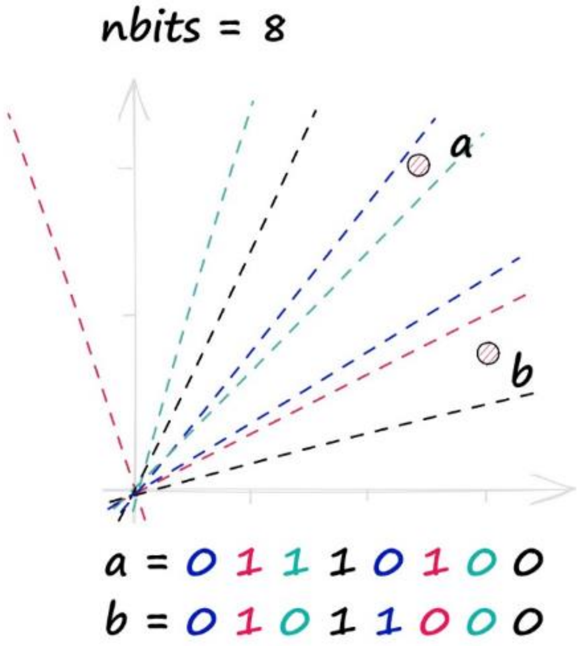
Locality-sensitive hashing: Random projection

Cool explanations: https://www.pinecone.io/learn/locality-sensitive-hashing-random-projection/

- We randomly sample n hyperplanes
- For each vector: bit=0 if on the left of the hyperplane, otherwise bit=1



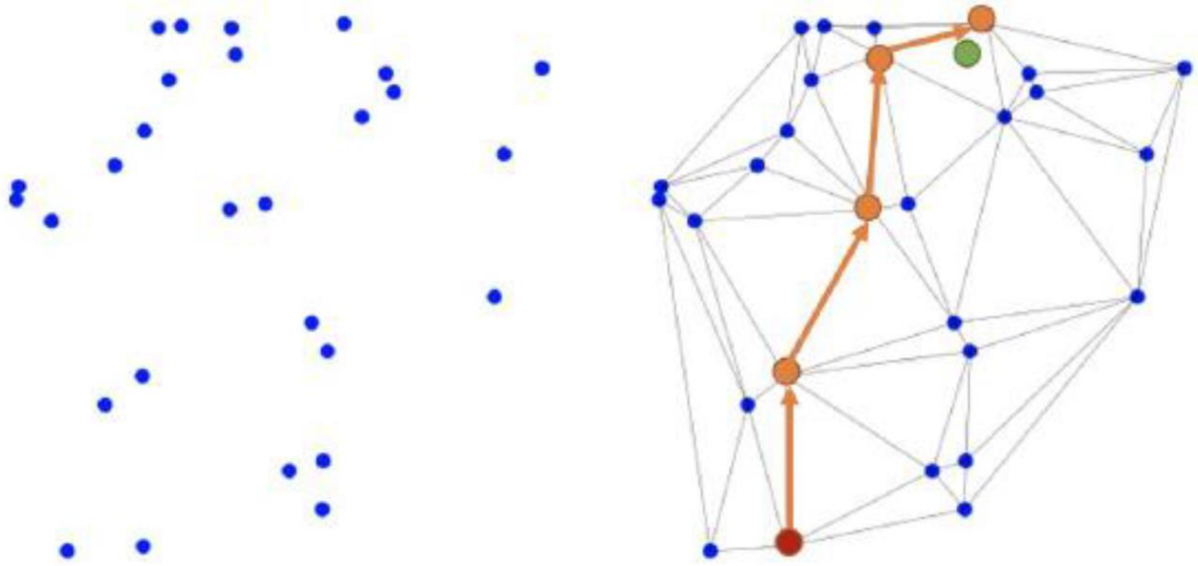
- We discretize the cosine similarity
- More bits = more accurate



4- Graph

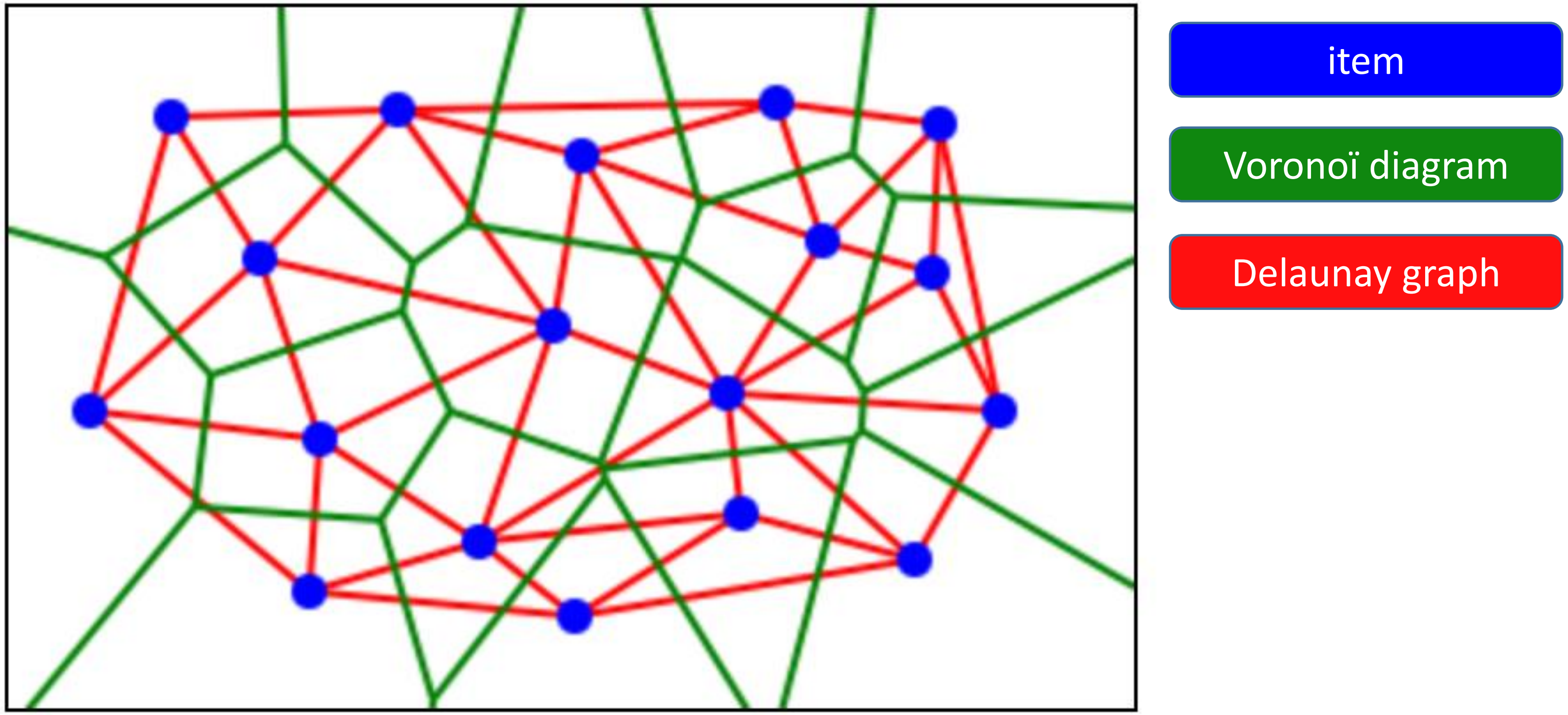
Graph-Based KNN

- The idea is to connect the vectors together in order to form a graph
- Then to search in the graph:
  - Initialize an empty list
  - Repeat n times:
    - Start from a random vector
    - Greedily navigate to reach the k closest vectors to the query vector in the graph
    - Update you list to keep the top k closest



But how to build such a graph???

Delaunay Graph (<1975)



All points within a Voronoï tile have contained blue point as their nearest neighbor.

Don’t expect to have Log(N) by navigating the graph naïvely ! Needs post processing

Does not wok in high dimension space ! (too many connections, too slow)

NSW (Navigable Small World)

« All people are six or fewer social connections from each other »

First manuscript written in ~1950 in Paris Sorbonne and circulated a lots. Milgram took up experiments 20 years later : take two random persons and see if/how the origin can manage to contact the destination.

➔ **Small world property exists in the nature !**

NSW (Navigable Small World)

How do we create small world network ?

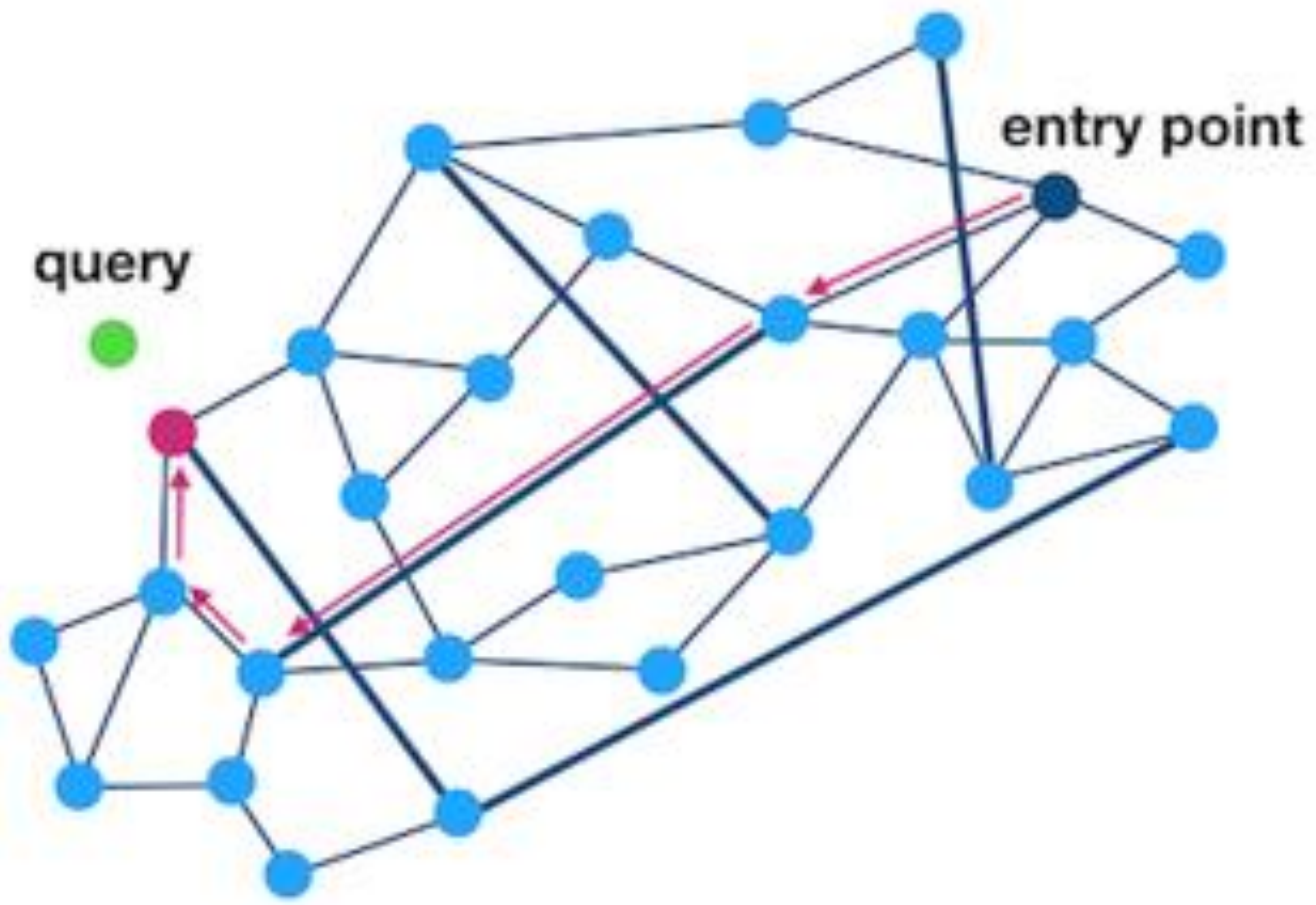
In social networks : Dual Phase Evolution

Local phase : people spend most of their time with people they know

Global phase : party/event/hollidays, where people interact with people they don't know

Links are created during global phases, refined/destroyed in local phase.

Small world network : is it enough ?



NSW (Navigable Small World)

Navigable property guarantees that greedy algorithm is likely to stay on the sortest path.

Construction : try to enforce every node is connected to its M nearest neighbors.

Search : BFS (piorize neighbors nearest to the query), with stop condition : Worst neighbor found is better than best candidate to visit.

Works not too bad but performance degradations in low dimension / clustered graph.

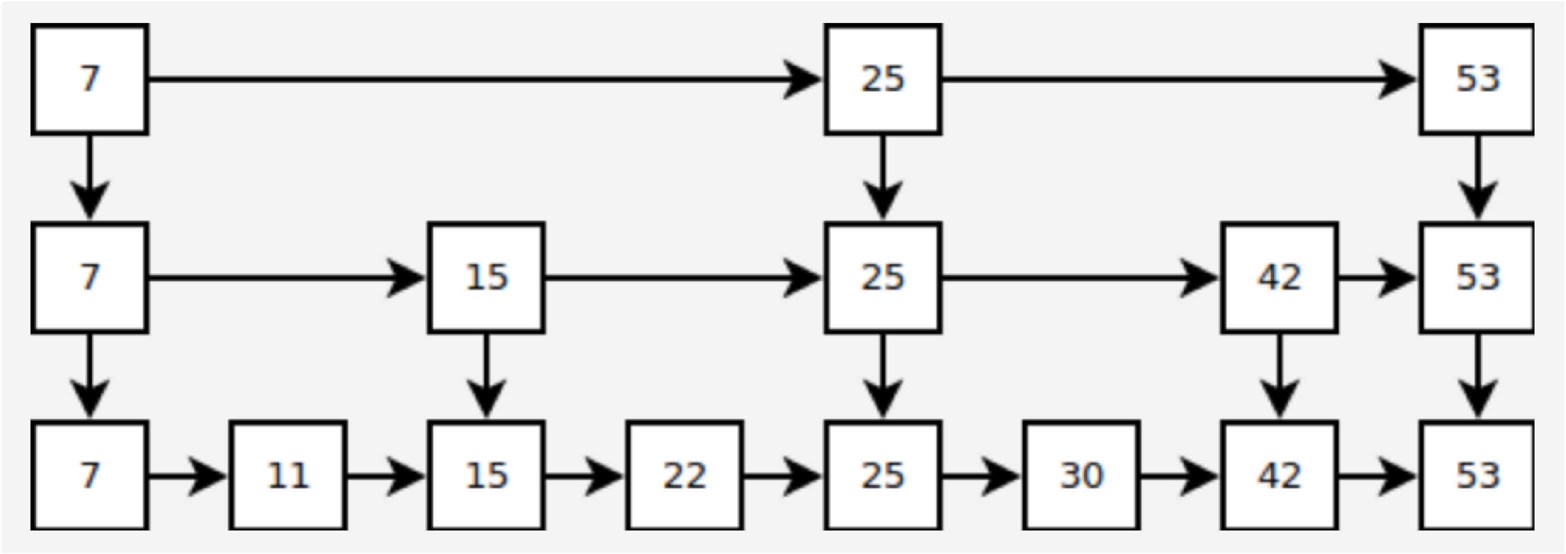
HNSW (Hierarchical NSW)

Idea

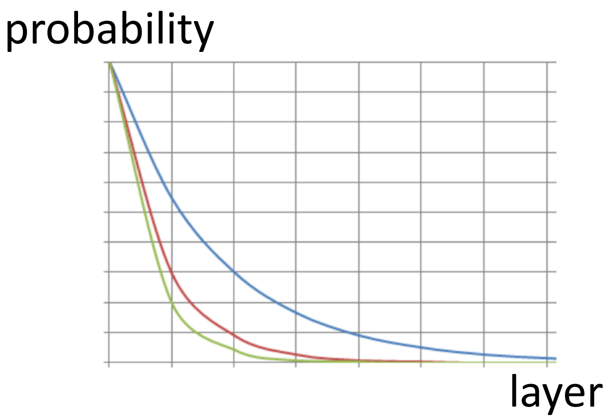
https://arxiv.org/ftp/arxiv/papers/1603/1603.09320.pdf

Add ‘highways’ to NSW

Same idea than skip-list and post-processing on Delaunay Graph !



HNSW - Construction



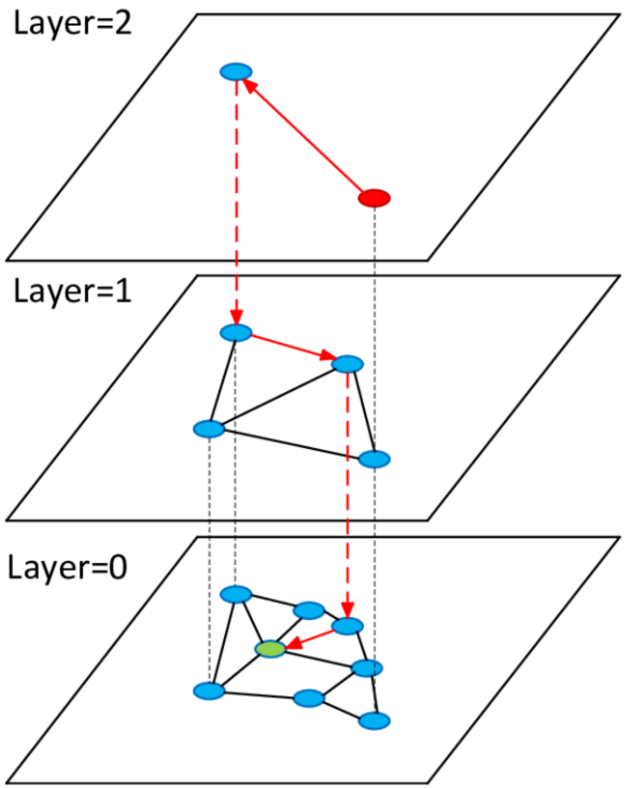
InsertNode(node, start)
Pick it's level L randomly (exponential decay law)
current 🡐 start
# search in upper layers
for layer in MaxL..L+1:
 current 🡐 KNNSearch(node, current, 1, layer)

# fill lower layers
for layer in L..0:
  neighbors 🡐 KNNSearch(node, current, M2, layer)
  toConnect 🡐 select(node, neighbors, M)
  for n in toConnect:
    connect node and n
    shrink connections of n
  current 🡐 neighbors

KNNSearch(query, start, k, layer)
//returns k closest neighbor to query

BFS from starting point, explore points nearest to query first.

Stop condition
Worst neighbor found is better than best candidate to visit.

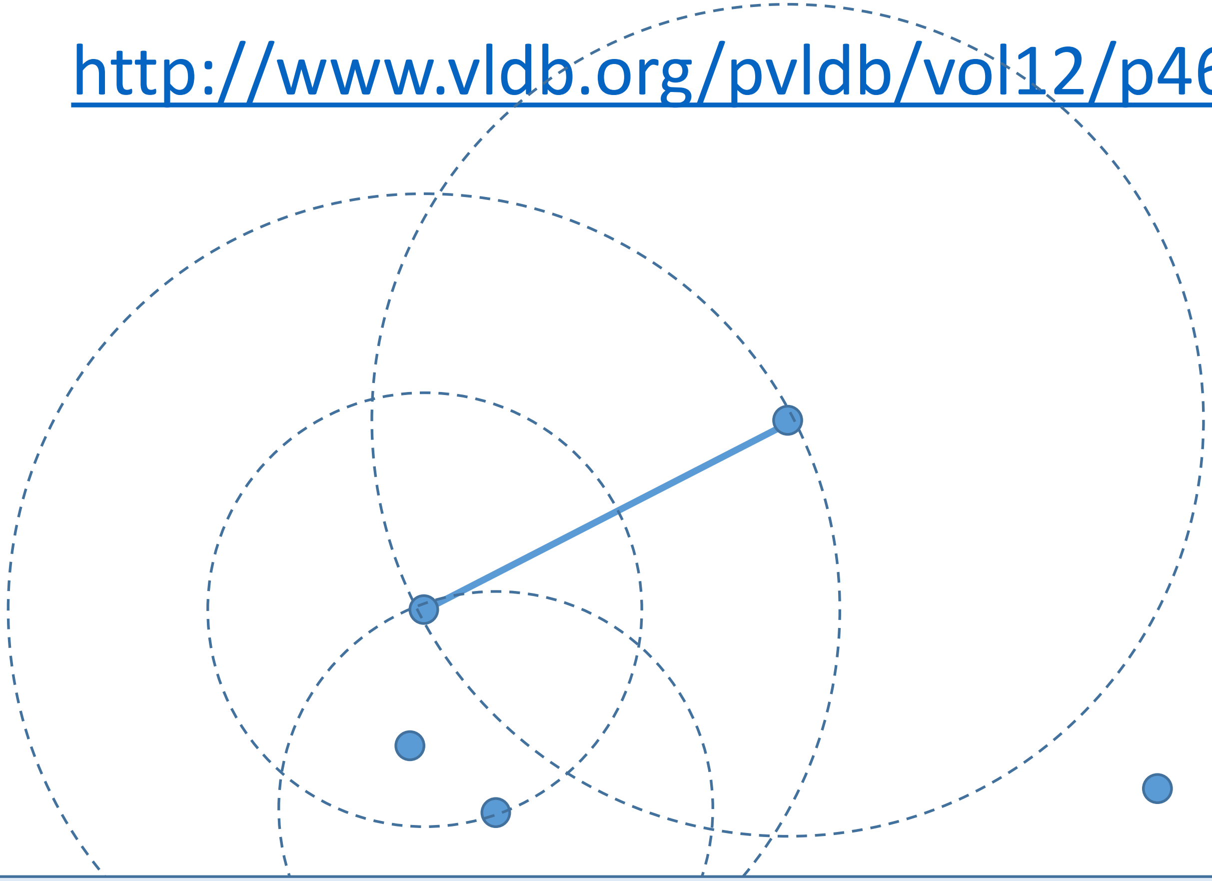


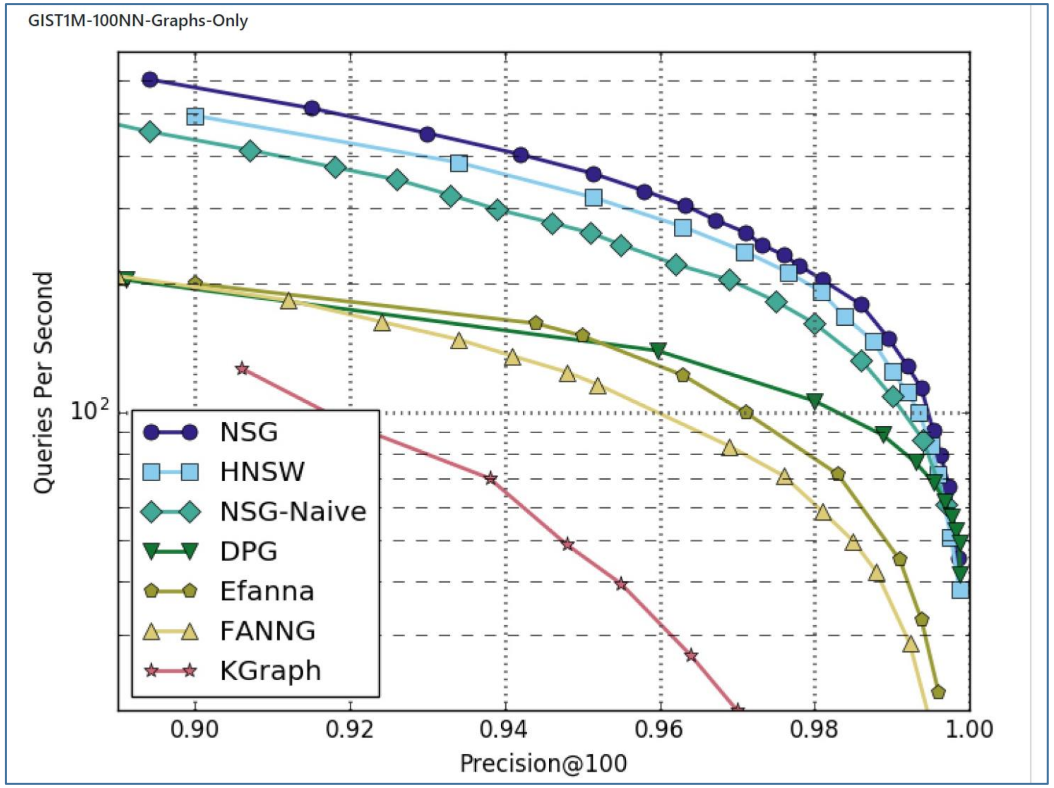
https://arxiv.org/ftp/arxiv/papers/1603/1603.09320.pdf

Still state of the art ?

EFANNA NSG (Alibaba)

http://www.vldb.org/pvldb/vol12/p461-fu.pdf





Connect p and q iif : $B(p, \delta(p,q)) \cap B(q, \delta(p,q)) \neq 0$

Still state of the art ?

NGT ONNG (yahoo Japan)

Same principle than hnsw with extra heuristics to decide which connections to cut when a node has too many connections.

https://arxiv.org/pdf/1810.07355.pdf

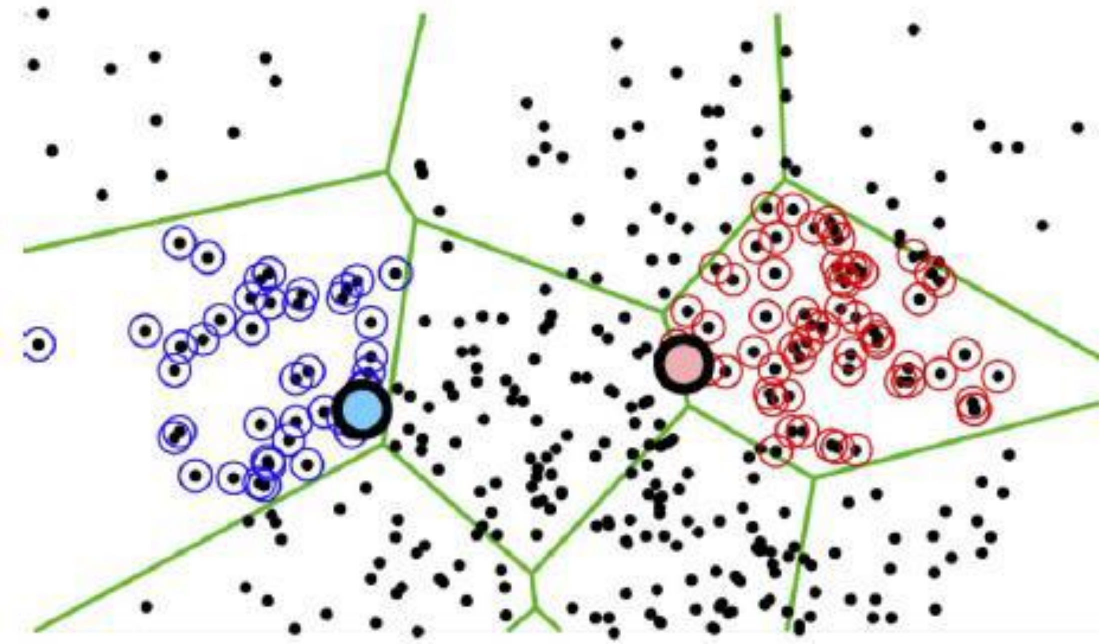
5 - IVF

Seine Saint Denis 93
Bobigny
Nanterre
Paris
Paris 75
Hauts de Seine 92
Créteil
Val de Marne 94
Lille
Pas de Calais 62
Nord 59
Arras
HAUTS-DE-FRANCE
Amiens
Somme 80
Seine Maritime 76
Rouen
Beauvais
Oise 60
Laon
Aisne 02
Charleville Mézières
Ardennes 08
Manche 50
Caen
NORMANDIE
Saint Lô
Calvados 14
Eure 27
Évreux
Pontoise
Val d'Oise 95
Seine
Marne
Chalons en Champagne
Meuse 55
Metz
Moselle 57
Rhin
Versailles
ÎLE-DE-FRANCE
Yvelines 78
Evry
Seine et Marne 77
Marne 51
Bar le Duc
Nancy
Meurthe et Moselle 54
Strasbourg
Bas Rhin 67
Orne 61
Essonne 91
Melun
GRAND-EST
Finistère 29
Saint Brieuc
Côtes d'Armor 22
Alençon
Chartres
Troyes
Épinal
Moselle
Quimper
BRETAGNE
Rennes
Ille et Vilaine 35
Mayenne 53
Sarthe 72
Eure et Loir 28
Aube 10
Chaumont
Vosges 88
Colmar
Morbihan 56
Vannes
Laval
Le Mans
Orléans
Loiret 45
Yonne 89
Haute Marne 52
Haut Rhin 68
Loir et Cher 41
Auxerre
Vesoul
Belfort
PAYS DE LA LOIRE
Loire Atlantique 44
Blois
Côte d'Or 21
Haute Saône 70
Territoire de Belfort 90
Angers
Tours
CENTRE-VAL-DE-LOIRE
BOURGOGNE-FRANCHE-COMTÉ
Dijon
Saône
Doubs
Besançon
Maine et Loire 49
Loire
Nantes
Indre et Loire 37
Bourges
Nièvre 58
Doubs 25
Nevers
Cher 18
Châteauroux
Deux Sèvres 79
La Roche sur Yon
Vendée 85
Poitiers
Vienne 86
Indre 36
Saône et Loire 71
Lons le Saunier
Jura 39
Moulins
Niort
Mâcon
Régions d'outre-mer :
971
Basse-Terre
GUADELOUPE
La Rochelle
Charente Maritime 17
Charente 16
Haute Vienne 87
Limoges
Guéret
Creuse 23
Allier 03
Puy de Dôme 63
Clermont Ferrand
Bourg en Bresse
Ain 01
Haute Savoie 74
Annecy
Rhône 69
Loire 42
Lyon
Saint Etienne
Chambéry
Savoie 73
Angoulême
Fort-de-France
972
MARTINIQUE
NOUVELLE-AQUITAINE
Périgueux
Corrèze 19
Tulle
AUVERGNE-RHÔNE-ALPES
Grenoble
Isère 38
Cantal 15
Haute Loire 43
Le Puy en Velay
Valence
Aurillac
Dordogne 24
Bordeaux
Dordogne
Cayenne
973
GUYANE FRANÇAISE
Gironde 33
Lot 46
Privas
Drôme 26
Hautes Alpes 05
Gap
Lot et Garonne 47
Cahors
Lot
Rodez
Mende
Lozère 48
Ardèche 07
Rhône
Landes 40
Agen
Tarn et Garonne 82
Aveyron 12
Gard 30
Vaucluse 84
Digne les Bains
Alpes de Haute Provence 04
Alpes Maritimes 06
Montauban
Albi
Avignon
Mont de Marsan
Saint-Denis
974
LA REUNION
Gers 32
Auch
OCCITANIE
Tarn 81
Nîmes
PROVENCE-ALPES-CÔTE D'AZUR
Nice
Bouches du Rhône 13
Toulouse
Montpellier
Hérault 34
Var 83
Marseille
Toulon
Pyrénées Atlantiques 64
Pau
Tarbes
Haute Garonne 31
Garonne
Carcassonne
Hautes Pyrénées 65
Foix
Ariège 09
Aude 11
976
Mamoudzou
MAYOTTE
Perpignan
Pyrénées Orientales 66
Bastia
Haute Corse 2B
CORSE
Ajaccio
Corse du Sud 2A

IVF (Inverted Flat index)

Index construction

- K-mean algorithm to find N clusters
- Put each vector in the corresponding cluster



Search time

- Find the closest cluster centers using an exhaustive index
- Then, explore all the points in the previously selected clusters and select the k closest to the input query.

Congrats! You now know the main categories of KNN indices!

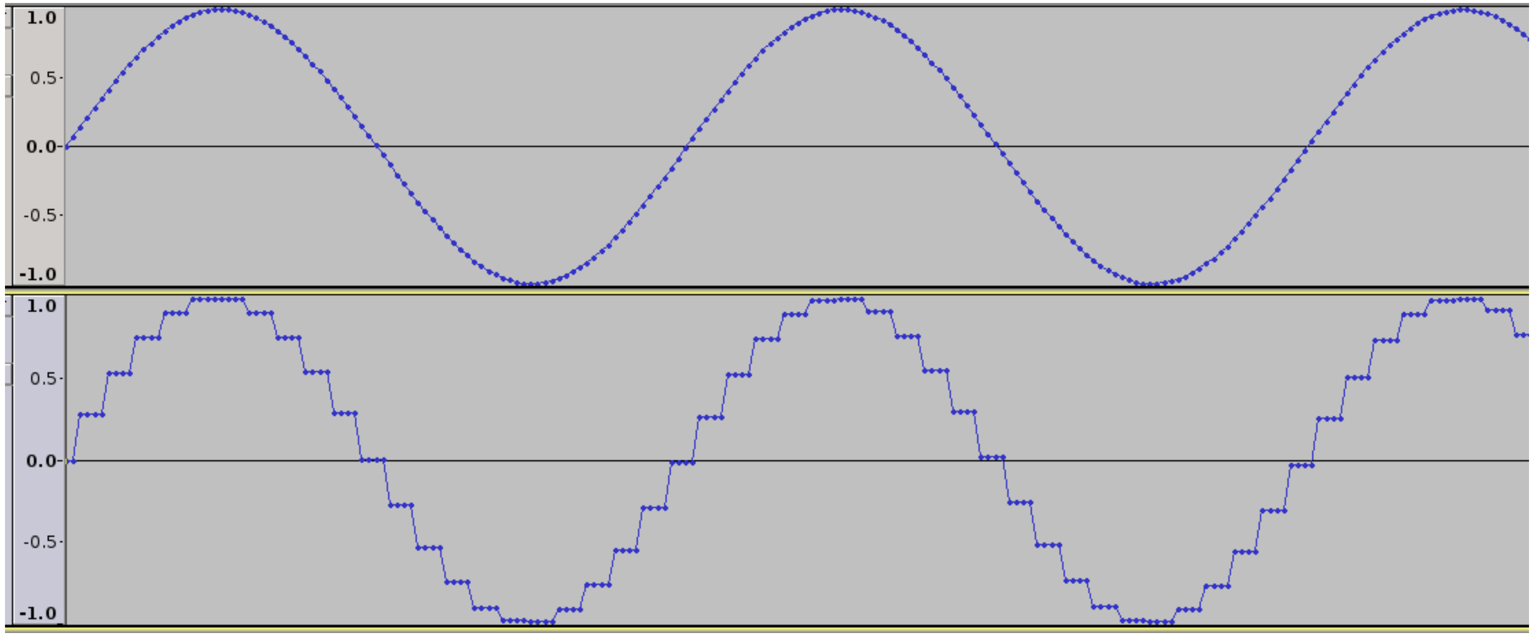
The bad news is that all these algorithms use too much RAM so scale to Billions of items!

	Brute Force	K-D trees	HNSW	Inverted index
Search time	Terrible	Great	Great	Good
KNN Score	Perfect	Terrible	Great	Great
Memory consumption	Same as input	+5-30% on input size	+5-30% on input size	+5-30% on input size
Index construction time	Instantaneous	Fast	Correct	Correct

6 – Quantized Indices

Scalar quantization

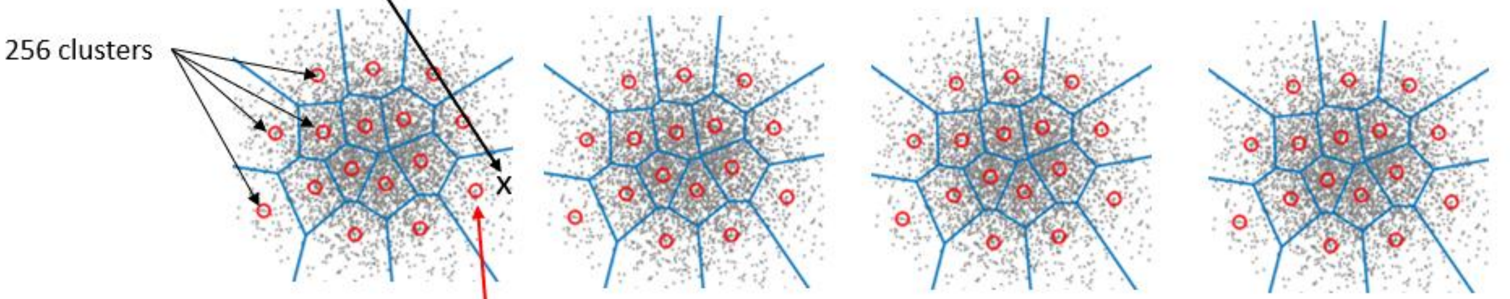
With scalar quantization you can **divide the memory space by a factor 2-4**



Product Quantization

Size in Bytes

Initial vector: $X = [x_1, x_2, x_3, x_4, x_5, x_6, x_7, x_8, x_9, x_{10}, x_{11}, x_{12}, x_{13}, x_{14}, x_{15}, x_{16}]$ — 16x4 = 64 Bytes

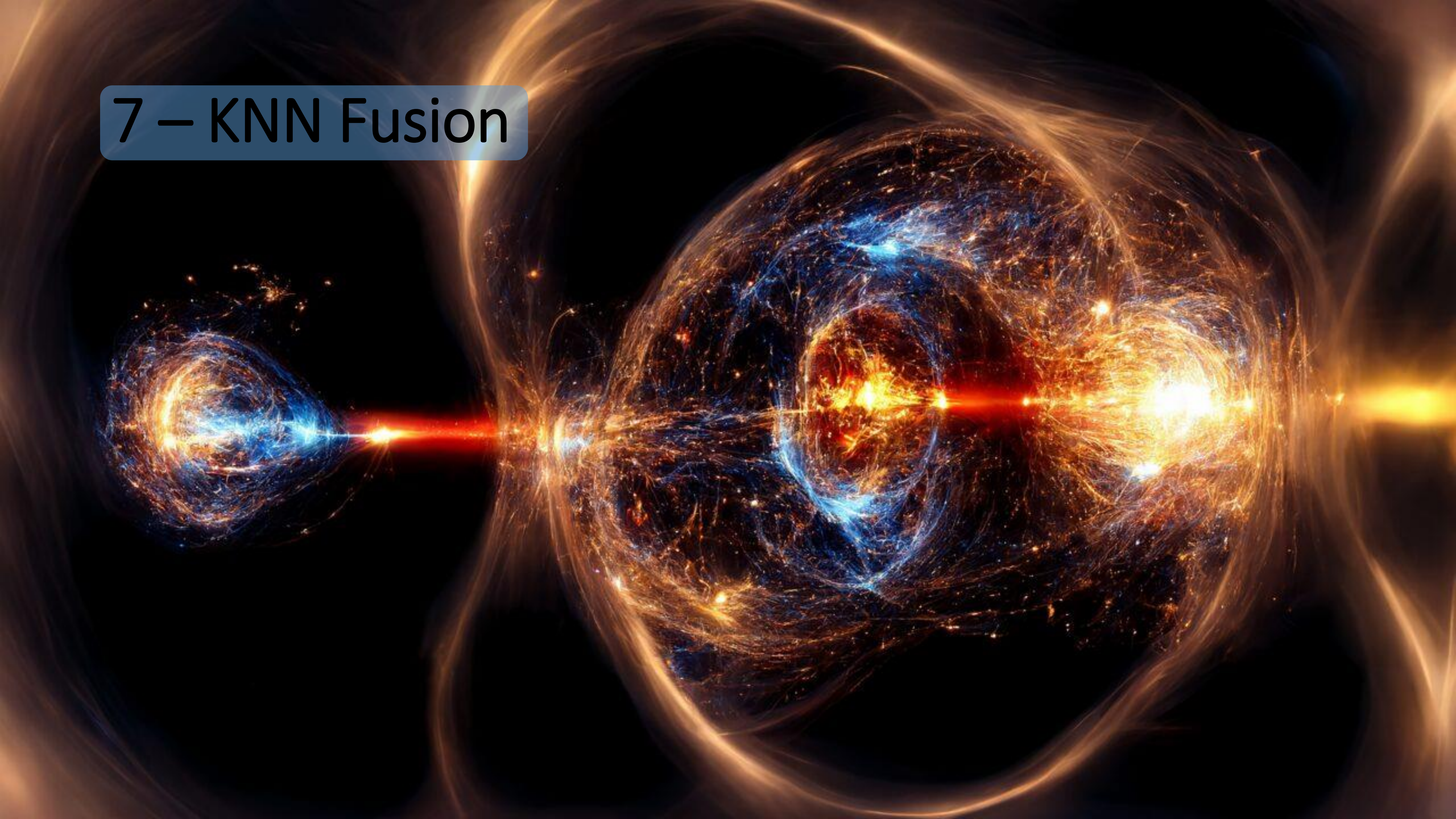


Closer to cluster c_1 Closer to cluster c_2 Closer to cluster c_3 Closer to cluster c_4

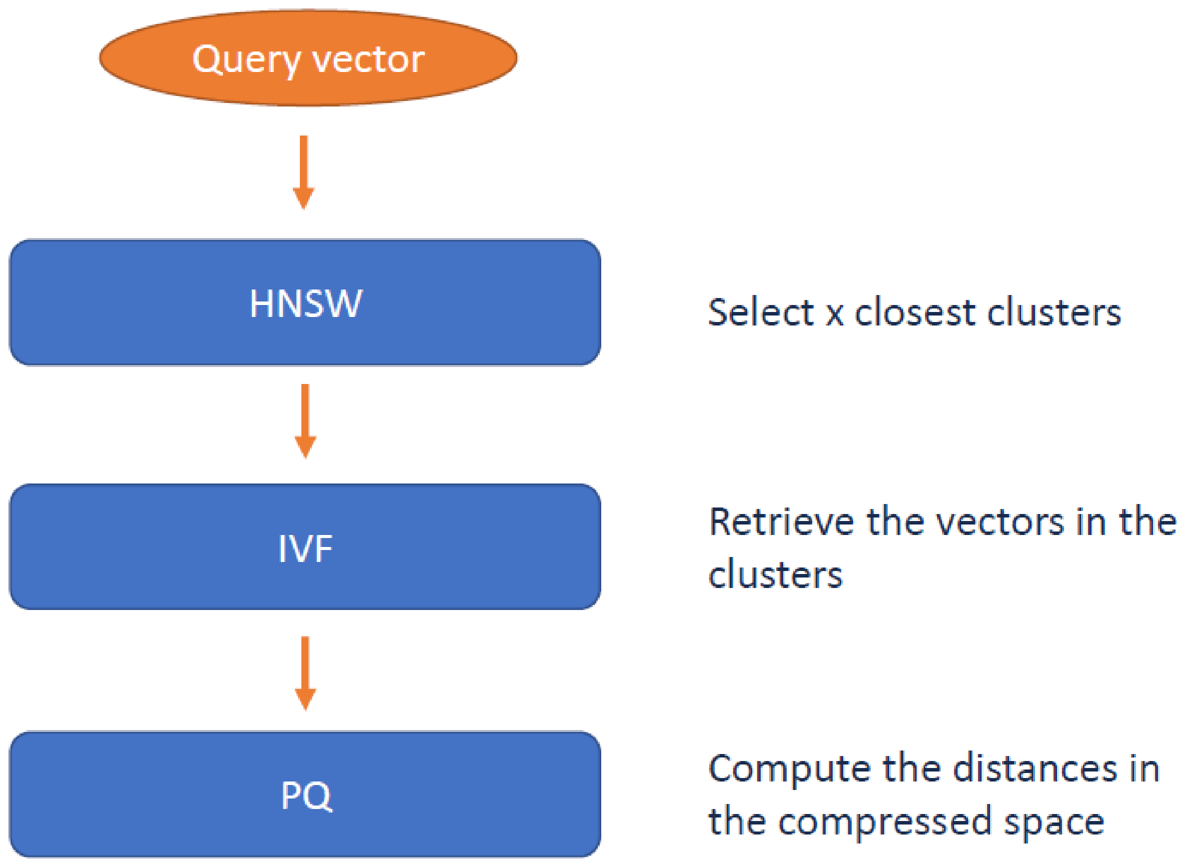
Stored vector: $[c_1, c_2, c_3, c_4]$ — 4 Bytes

Reconst vector: $X = [c_{1,1}, c_{2,2}, c_{3,3}, c_{4,4}, c_{5,1}, c_{6,2}, c_{7,3}, c_{8,4}, c_{9,1}, c_{10,2}, c_{11,3}, c_{12,4}, c_{13,1}, c_{14,2}, c_{15,3}, c_{16,4}]$ — 64 Bytes

7 – KNN Fusion



OPQ64_256,IVF131072_HNSW32,PQ64x8-nprobe=18,efSearch=36,ht=2048.index



8- KNN in practice!

KNN in practice!

Faiss: a library implementing very efficiently the most common indexing algorithms

- C++ lib with many bindings (ex: python)
- Github: https://github.com/facebookresearch/faiss

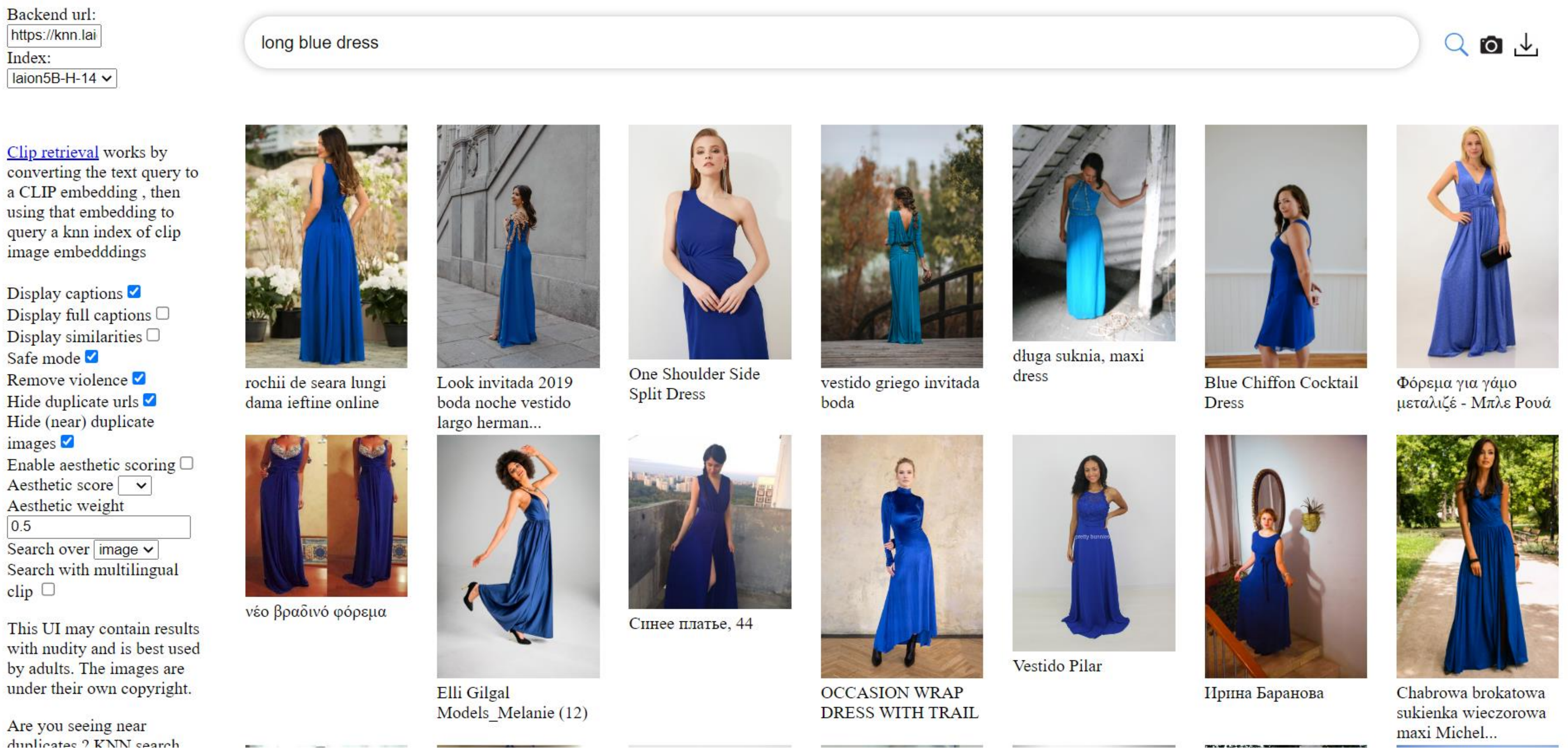
Hnswlib: Opensource library for HNSW, Criteo contributed actively on it

- C++ lib with python and Java bindings
- Github: https://github.com/nmslib/hnswlib

Autofaiss: Opensource python library to automatically create KNN indices from your dataset (Can create in few hours)

- Wrapper library on top of Faiss to simplify its usage
- It is possible to build a large (400 million vectors, 2TB) KNN index in 3 hours -in a **low amount of memory (16 GB)** with **latency in milliseconds (10ms)**
- Medium article: https://medium.com/criteo-engineering/introducing-autofaiss-an-automatic-k-nearest-neighbor-indexing-library-at-scale-c90842005a11
- Github: https://github.com/criteo/autofaiss

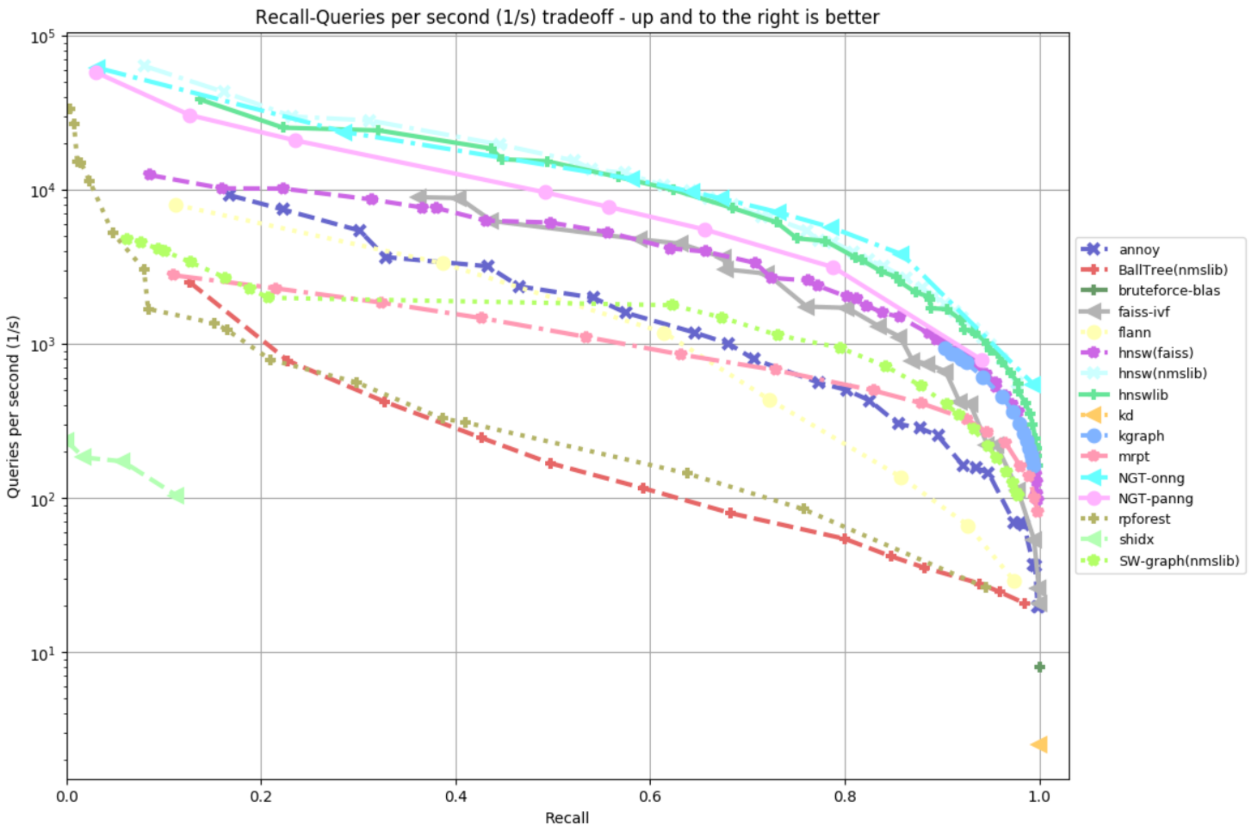
Real life example



https://rom1504.github.io/clip-retrieval/?back=https%3A%2F%2Fknn.laion.ai&index=laion5B-H-14&useMclip=false

Benchmark Results

glove-100-angular



Current research in the domain

- Hybrid KNN (DiskANN, SPANN, …)
- Optimization for inner product (QUIPS, SCANN, …)
- Compressed embeddings at training time (ROBE-Z, SCMA, …)
- NeurIPS’21: Billion-Scale Approximate Nearest Neighbor Search Challenge (http://big-ann-benchmarks.com)

Time to practice!